EXAM
Advanced Functional Programming
TDA340/INN260

DAY: 19/12 2006 TIME: 8.30 – 12.30 PLACE: V-huset

Responsible:	Koen Lindström Claessen, 772 5424
Result:	Announced latest January 10th
Aids:	None
Grade:	At least 30p are required to pass

Note the following:

- Answers can be given in English or Swedish
- Put page numbers on each paper sheet
- Start each assignment on a new page
- Write your personal number on all pages
- Please read every assignment **carefully** before you answer
- Write clearly; unreadable = wrong!
- Less points are given for unnecessarily long-winded, complicated, or unstructured solutions
- Write down clearly if you make assumptions that are not mentioned in the assignment

Good luck!

Assignment 1 — Monads (20p)

(a) Explain the concept of a monad in Haskell, what it usually models, and what its typical applications are.

(b) What is the advantage of capturing the monad concept in a type class?

(c) Give three examples of "standard" monads in Haskell. For each monad, explain what it does, give its type declaration, list its special operations and their implementation, and give its monad type class instance.

(d) For one such monad, give a (small) example (using Haskell code) of how it can be used. Explain what your example does.

(e) Describe the three *monad laws*, that relate the functions `(>>=)` and `return`.

(f) For each law, explain what it says, and also whay this is a reasonable thing to require of a monad.

Assignment 2 — Monad Transformers (20p)

(a) Explain the concept of a monad transformer. What activity do monad transformers enable that regular monads do not enable?

(b) Describe the implementation of the *state monad transformer*, adding the feature of an updatable state to any monad. Assume the name of the type is `StateT s m a`, where `s` is the type of the state, and `m` is the underlying monad. Give the type declaration, and the implementations of its special operations, its monad type class instance, and its lift function.

Take a look at the following datatype declaration for simplified arithmetic *C-expressions*. In C, apart from talking about numbers, an arithmetic expression can have a side effect, such as assigning a new value to a variable, or printing a string on the screen.

```
type Var = String

data CExpr
  = Num Int
  | Add CExpr CExpr
  | Var Var
  | Var := CExpr
  | Print String
 deriving ( Show )
```

In order to be able to deal with variables and values, we define the concept of a *store*, that keeps track of the values of variables.

```
type Store = [(Var,Int)]
```

When we evaluate a C-expression, we need to know the values of the variables. As a result, we produce the final answer, together with the updated store, and the output that was produced. Evaluating an expression can go wrong as well, for example when a variable is used without being in the store. Thus, we end up with the following evaluation function.

```
eval :: Store -> CExpr -> Maybe (Int, Store, [String])
eval st (Num n)   = Just (n, st, [])
eval st (Add a b) = case eval st a of
                      Just (n, st1, o1) ->
                        case eval st1 b of
                          Just (m, st2, o2) ->
                            Just (n+m, st2, o1++o2)
                          _ -> Nothing
                      _ -> Nothing
eval st (Var v)   = case lookup v st of
                      Just n  -> Just (n, st, [])
                      Nothing -> Nothing
eval st (x := a)  = case eval st a of
                      Just (n, st', o) -> Just (n, (x,n):st', o)
eval st (Print s) = Just (0, st, [s])
```

(c) Which three distinct monads can you identify in the above?

(d) Define a monad that the above `eval` function can use, and re-implement the `eval` function using it.
You are allowed to use standard monad transformers, but you do not have to.

Assignment 3 — Embedded Languages (20p)

In this assignment, you will think about the design and implementation of an embedded language for ASCII art. As always, the embedded language should be compositional, i.e. we want to create large complex images out of small simple images.

Here is an example of something we would like to describe:

```
+---+---+---+   +-----+
| A | B | C +---+ apa |
+---+---+---+   +--+--+
                   |
                 +-+-+
                 | f |
                 | o |
                 | o |
                 +---+
```

Typical things you would like in your pictures are: boxes, horizontal lines, vertical lines, horizontal text, vertical text, etc.

(a) Design an API for the above embedded language. This should consist of: appropriate names of types, and names and types of creation functions, combination functions, and run functions. For each function, describe briefly what it is supposed to do.
Here, try to keep the role of each function as simple as possible, but add enough combinators to allow to describe the picture above, for example.

(b) Which of the functions in your API are primitive (i.e. you can express the others in terms of the primitive ones)? Give the definitions of the non-primitive functions in terms of the primitive functions.

(c) What are the advantages of embedding your API as a library over inventing your own language, and have your program read in image descriptions from a text file? Mention advantages from the programmer's (= you) point of view, as well as the user's point of view.

(d) What properties (or laws) do your functions have? Mention at least three non-trivial such ways in which your functions interact.

(e) Describe what a *shallow* implementation (i.e. you implement ASCII images semantically) could look like. Give a type definition, and describe (in words) what each of your primitive functions, as well as your run function, would do.

(f) Describe what a *deep* implementation (i.e. you implement ASCII images by the way they were created) could look like. Give a type definition, and describe (in words) what each of your primitive functions, as well as your run function, would do.

(g) Discuss at least one advantage for each of the two implementation approaches.

(total 60p)